Efficient Strategies for Generating Collisions in SHA-2/SHA-3 Outputs

25 August 2022
Simon Edwards
Research Acceleration Initiative

Introduction

Cryptographic hashing, mysterious as it may be to most, provides security on the basis of the assumption that an input cannot be extrapolated from an output and that specific outputs cannot be duplicated or anticipated by controlling inputs. Ideally, hash outputs should be uniform in terms of the statistical distribution of values in outputs, but that is actually the easy part. Creating a system in which collisions cannot be generated is the first, second, and third priority in such an endeavor.

Abstract

Secure Hashing Algorithms operate on the basis of processing large chunks of data, usually on a word by word basis, with each word being composed of four bytes (although word size may vary between algorithms.) These words then have their sequence altered through a series or "switch boxes" in which, for example, the first character in the word might be switched with the fourth and the second with the third. The modified word is then "inflated" using a very large prime exponent which is then truncated to its final 64 bytes and added to the digest, but only after those output bytes are themselves put through a series of switch boxes that scramble what is added to the digest. The modified digest value of the second word, 64 bytes in length, is added to the 64-byte value associated with the first word and this combined number, generally 65 bytes long, is truncated.

You would think that the first byte of the second-round digest output would always be added to the first byte of the previous, and so on. Unfortunately, this is not how the National Security Agency designed the algorithm. Consequently, even the SHA-3 algorithm which was independently developed to provide an alternative in the event SHA-2 was compromised follows this same principle of adding the whole 64-byte output to the previous as a whole number and truncating that number, a number far too small for there not to be patterns created since only one byte is being truncated each time.

Thus, when you regard the way in which SHA-2/3 handles switch boxes and inflation during the processing of each individual word, there are no problems. Again, no problems with truncating a number that is on the scale of FFBB7799^9786013, for instance. The problem comes in when you add two numbers that are each only 64 bytes long to one another and keep truncating.

To demonstrate this, I invite the community to attempt to generate collisions in the following way:

Start with strings of exactly twenty bytes that are all sequential, all of the letters from A-T for instance, and select two sets of two of those characters in which each of the two characters being inverted sit on the boundary between

two words (in this case, it would have to be word one and two as well as word four and five.) Invert the sequence of those two characters with respect only to each other (but not the sets as a whole with the other set) and then select at random a single character from the only remaining word (word three) not affected by the inversion and delete that. Hash the result and compare to the original sequence [ABCD|EFGH|IJKL|MNOP|QRST] VS. [ABCE|DGHI|_KLM|NOQ|PRST] and repeat every possible permutation of inversions and deletions that accord with the rules laid out. Please note that after deleting a character from word three, the boundary between words four and five would change. We are inverting the last letter in word four and five PRIOR TO making the deletion of a random character in word three, to be clear.

In any given twenty-byte test sequence, there are only four possible combinations to try. Once complete, try the next sequence (B-Q) and so on. There will be many to try since fundamentally we are dealing with hex characters 00-FF and when all of those combinations have been tried, one can try 21-byte strings. The important thing is that the last letter of the first word and and first letter of the second word are alternated and that the same is done at the boundary of the second to last word and the last word.

The reason why I believe you will find this approach particularly effective is because the act of alternating the position of neighboring bytes in two different places creates a greater chance of a collision in higher and lower place values simultaneously, and since we're dealing with truncation (and we don't know whether the next sequential byte will increase or decrease the digest output due to the size of the exponent) we can at least be sure that if we kick a byte value into a preceding word, we are making sure that we are ultimately adding that value, regardless of where it lands within the word after switch boxing, to the same value it was being added to the first time around. Consequently, no matter how large the exponent, similar numbers are being added together and whether we add A+B or we add B+A, we are duplicating the conditions that led to the algorithm spitting out a given value since we've in a single fell swoop pre-empted a switching step and made sure that some of the same values are ultimately added. The odds of a collision are then further amplified through the aforementioned deletion step due to the way in which place values are shifted. The mathematical reason behind this tendency is very similar to the reason why dividing a value of one by an odd number tends to create repeating sequences. Thus, this approach creates patterns in multiple dimensions (un-switching, place value shifting through deletion, and byte switching at word boundaries.)

Taking this a step further, after all possible sequential strings up to 64 bytes in length are tried, the next most likely approach to increase the statistical likelihood of generating collisions lies in hashing a hybrid string consisting of the original string followed by the altered string (or the altered string followed by the original string.) Once all of those combinations have been tried (if a collision hasn't already been found) then collisions can be found by fractalization, which I define as duplicating segments of strings already tried and inserting them at random anywhere within the string, including at the beginning or end.

These approaches can be programmed readily by any reasonably adept

computer scientist/programmer and a brute force approach guided by these rules can be used to verify the strategy.

Conclusion

A top-down re-evaluation of SHA design is called for as any system which relies upon steps that include switch boxes and truncations would be vulnerable to this sort of collision attack that employs this particular combination of byte reversals and deletions strategically. New approaches including the use of switching between multiple exponents from one word to the next and variable word sizes dictated by a hidden function would do more to improve overall strength than switch boxes, which if my hypothesis is correct, actually introduce weakness to an otherwise strong system.